4.1 Operator Overloading

Unit 4: Advanced Class Features

Object Oriented Programming with C++

Unit 4: Advanced Class Features - Operator Overloading

- Lecture 29: 4.1 Operator Overloading
- Topics:
- - Unary and Binary Operators
- - Member and Friend Functions
- - Rules and Limitations

Unit 4: Advanced Class Features - Operator Overloading
- Lecture 29: 4.1 Operator Overloading
- Topics:
- - Unary and Binary Operators
- - Member and Friend Functions
- - Rules and Limitations

Welcome class! Today we begin Unit 4 by diving into one of the most powerful and expressive features of C++: Operator Overloading. We will learn how to make our custom objects interact seamlessly with standard C++ operators like plus, minus, and equals.

What is Operator Overloading?

- A type of compile-time polymorphism in C++.
- Allows developers to redefine the way standard operators work when applied to user-defined data types (objects).
- Provides a special, context-specific meaning to an existing operator.
- Does NOT change the meaning of the operator for built-in fundamental types (like int, float).

What is Operator Overloading?
- A type of compile-time polymorphism in C++.
- Allows developers to redefine the way standard operators work when applied to user-defined data types (objects).
- Provides a special, context-specific meaning to an existing operator.
- Does NOT change the meaning of the operator for built-in fundamental types (like int, float).

In C++, we know we can overload functions—giving the same name to multiple functions with different parameters. Similarly, C++ allows us to overload operators. This means we can teach our custom classes how to behave when a standard operator is applied to them. Importantly, this is compile-time polymorphism, and it doesn't break how operators work for standard types like integers.

Why Use Operator Overloading?

- Improves code readability and expressiveness.
- Allows user-defined classes to behave similarly to primitive built-in types.
- Example without overloading: c3 = c1.add(c2);
- Example with overloading: c3 = c1 + c2;
- Reduces cognitive load when dealing with mathematical or logical abstractions like Matrices, Vectors, and Complex Numbers.

Why Use Operator Overloading?
- Improves code readability and expressiveness.
- Allows user-defined classes to behave similarly to primitive built-in types.
- Example without overloading: c3 = c1.add(c2);
- Example with overloading: c3 = c1 + c2;
- Reduces cognitive load when dealing with mathematical or logical abstractions like Matrices, Vectors, and Complex Numbers.

Why do we need this? Think about readability. If you have two complex numbers, c1 and c2, writing `c1 + c2` is mathematically intuitive. Writing `c1.add(c2)` is clunky and harder to read, especially in complex equations. Operator overloading brings our custom types up to the same level of convenience as native types.

General Syntax of Operator Overloading

- Achieved using a special function called an 'operator function'.
- Keyword: `operator`
- Syntax: `return_type operator op (argument_list) { ... }`
- - `return_type`: Data type of the value returned by the operation.
- - `operator`: Keyword indicating an overloaded operator.
- - `op`: The specific symbol being overloaded (e.g., +, -, *).
- - `argument_list`: Operands passed to the operator function.

General Syntax of Operator Overloading
- Achieved using a special function called an 'operator function'.
- Keyword: operator
- Syntax: return_type operator op (argument_list) { ... }
- - return_type: Data type of the value returned by the operation.
- - operator: Keyword indicating an overloaded operator.
- - op: The specific symbol being overloaded (e.g., +, -, *).
- - argument_list: Operands passed to the operator function.

To overload an operator, we declare a special function. The name of this function always consists of the keyword 'operator' followed by the symbol we want to overload, such as 'operator+'. The return type depends on what the operation should yield, and the argument list depends on the number of operands.

Rules for Operator Overloading

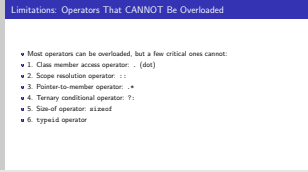
- 1. Only existing C++ operators can be overloaded. You cannot invent new operators (e.g., for power).
- 2. At least one operand must be of a user-defined type.
- 3. You cannot change the fundamental semantics (e.g., you shouldn't make + perform subtraction).
- 4. Precedence, associativity, and the number of operands of the original operator cannot be changed.
- 5. Default arguments cannot be used in operator functions.

Rules for Operator Overloading
- 1. Only existing C++ operators can be overloaded. You cannot invent new operators (e.g., for power).
- 2. At least one operand must be of a user-defined type.
- 3. You cannot change the fundamental semantics (e.g., you shouldn't make + perform subtraction).
- 4. Precedence, associativity, and the number of operands of the original operator cannot be changed.
- 5. Default arguments cannot be used in operator functions.

The compiler enforces strict rules. You cannot invent new operator symbols. You cannot overload operators exclusively for primitive types—at least one operand must be an object. While C++ lets you do anything inside the function block, it's a terrible practice to change the mathematical meaning of an operator. Finally, precedence and associativity remain completely unaffected.

Limitations: Operators That CANNOT Be Overloaded

- Most operators can be overloaded, but a few critical ones cannot:
- 1. Class member access operator: `.` (dot)
- 2. Scope resolution operator: `::`
- 3. Pointer-to-member operator: `.*`
- 4. Ternary conditional operator: `?:`
- 5. Size-of operator: `sizeof`
- 6. `typeid` operator



Despite its flexibility, C++ protects certain fundamental operators. The dot operator, scope resolution operator, pointer-to-member, sizeof, and the ternary conditional operator cannot be overloaded. These are deeply integrated into the compiler's core mechanics and object memory representation, so altering them would break the language structurally.

Unary vs. Binary Operators

- Unary Operators:
- - Operate on a single operand.
- - Examples: ++ (increment), -- (decrement), - (unary minus, negation), ! (logical NOT).
- Binary Operators:
- - Operate on two operands (left and right).
- - Examples: + (addition), - (subtraction), * (multiplication), == (equality).

Unary vs. Binary Operators
- Unary Operators:
- - Operate on a single operand.
- - Examples: ++ (increment), -- (decrement), - (unary minus, negation), ! (logical NOT).
- Binary Operators:
- - Operate on two operands (left and right).
- - Examples: + (addition), - (subtraction), * (multiplication), == (equality).

Operators are generally classified by how many operands they take. Unary operators need only one operand, like changing the sign of a number with a unary minus. Binary operators require two operands, a left side and a right side. The function signatures for overloading these two types differ significantly.

Overloading Unary Operators (Member Function)

- When overloaded as a member function, a unary operator takes NO arguments.
- The object invoking the function is implicitly passed via the hidden `this` pointer.
- Syntax within class:
- `return_type operator op();`
- Example call: `-obj;` is translated to `obj.operator-();`

Overloading Unary Operators (Member Function)
- When overloaded as a member function, a unary operator takes NO arguments.
- The object invoking the function is implicitly passed via the hidden this pointer.
- Syntax within class:
- return_type operator op();
- Example call: -obj; is translated to obj.operator-();

Let's start with unary operators implemented as member functions. Because a unary operator acts on a single object, and a member function is called explicitly on an object, the object itself is the operand. Therefore, the function requires zero arguments; the compiler automatically provides access to the object via the 'this' pointer.

Example: Unary Minus Operator (Member Function)

- class Point {
- int x, y;
- public:
- Point(int x, int y) : x(x), y(y) {}
- void operator-() { // No arguments
- x = -x;
- y = -y;
- }
- };
- int main() {
- Point p1(5, 10);
- -p1; // Negates coordinates
- }

Example: Unary Minus Operator (Member Function)
class Point {
int x, y;
public:
Point(int x, int y) : x(x), y(y) {}
void operator-() { // No arguments
x = -x;
y = -y;
}
};
int main() {
Point p1(5, 10);
-p1; // Negates coordinates
}

Here is a concrete example. We have a Point class. Inside, we declare `void operator-()`. Notice there are no parameters. When we call `-p1` in main, C++ translates this to `p1.operator-()`. The x and y values of p1 are negated directly within the function block.

Overloading Unary Operators (Friend Function)

- When overloaded as a friend function, a unary operator takes EXACTLY ONE argument.
- Friend functions do not have a `this` pointer, so the operand must be passed explicitly.
- Syntax within class:
- `friend return_type operator op(ClassType& obj);`
- Passing by reference allows modification of the original object.

Overloading Unary Operators (Friend Function)
- When overloaded as a friend function, a unary operator takes EXACTLY ONE argument.
- Friend functions do not have a this pointer, so the operand must be passed explicitly.
- Syntax within class:
- friend return_type operator op(ClassType& obj);
- Passing by reference allows modification of the original object.

What if we use a friend function? Friend functions are not members, so they lack a 'this' pointer. Thus, we must explicitly pass the object we want to operate on. A unary operator overloaded as a friend function takes exactly one argument, usually passed by reference so the function can alter the original object's state.

Example: Unary Minus Operator (Friend Function)

- class Point {
- int x, y;
- public:
- Point(int x, int y) : x(x), y(y) {}
- friend void operator-(Point& p);
- };
- void operator-(Point& p) {
- p.x = -p.x;
- p.y = -p.y;
- }
- // main: Point p1(5, 10); -p1; calls operator-(p1);

Example: Unary Minus Operator (Friend Function)
class Point {
int x, y;
public:
Point(int x, int y) : x(x), y(y) {}
friend void operator-(Point& p);
};
void operator-(Point& p) {
p.x = -p.x;
p.y = -p.y;
}
// main: Point p1(5, 10); -p1; calls operator-(p1);

Looking at the same Point class, we now use a friend function. We declare it inside with the 'friend' keyword and define it outside. The function `operator-` takes a `Point&` as its single parameter. When the compiler evaluates `-p1`, it effectively calls `operator-(p1)`.

Overloading Binary Operators (Member Function)

- When overloaded as a member function, a binary operator takes EXACTLY ONE argument.
- The left-hand operand is the calling object (implicitly passed via `this`).
- The right-hand operand is explicitly passed as the function's argument.
- Syntax within class:
- `return_type operator op(const ClassType& rhs);`
- Example call: `A + B` translates to `A.operator+(B)`

Overloading Binary Operators (Member Function)
- When overloaded as a member function, a binary operator takes EXACTLY ONE argument.
- The left-hand operand is the calling object (implicitly passed via this).
- The right-hand operand is explicitly passed as the function's argument.
- Syntax within class:
- return_type operator op(const ClassType& rhs);
- Example call: A + B translates to A.operator+(B)

Moving to binary operators which require two operands. When using a member function, the left-hand operand is the object invoking the method. The right-hand operand is passed as the single explicit parameter. So, an expression like `A + B` evaluates as `A.operator+(B)`.

Example: Binary Plus (+) Operator (Member Function)

- class Complex {
- int real, imag;
- public:
- Complex(int r=0, int i=0) : real(r), imag(i) {}
- Complex operator+(const Complex& rhs) {
- Complex temp;
- temp.real = real + rhs.real;
- temp.imag = imag + rhs.imag;
- return temp;
- }
- };
- // main: Complex c3 = c1 + c2;

Example: Binary Plus (+) Operator (Member Function)
class Complex {
int real, imag;
public:
Complex(int r=0, int i=0) : real(r), imag(i) {}
Complex operator+(const Complex& rhs) {
Complex temp;
temp.real = real + rhs.real;
temp.imag = imag + rhs.imag;
return temp;
}
};
// main: Complex c3 = c1 + c2;

In this Complex number example, we overload the + operator. It takes one `Complex` object as a parameter (`rhs`). `real` refers to the left operand's attribute, and `rhs.real` refers to the right operand. The function creates a new `Complex` object `temp`, computes the sum, and returns it by value.

Overloading Binary Operators (Friend Function)

- When overloaded as a friend function, a binary operator takes EXACTLY TWO arguments.
- Because there is no `this` pointer, both the left and right operands must be provided.
- Syntax within class:
- `friend return_type operator op(const ClassType& lhs, const ClassType& rhs);`
- Example call: `A + B` translates to `operator+(A, B)`

Overloading Binary Operators (Friend Function)
- When overloaded as a friend function, a binary operator takes EXACTLY TWO arguments.
- Because there is no this pointer, both the left and right operands must be provided.
- Syntax within class:
- friend return_type operator op(const ClassType& lhs, const ClassType& rhs);
- Example call: A + B translates to operator+(A, B)

If we implement a binary operator as a friend function, we must supply both operands as arguments because there is no calling object. Therefore, a binary operator as a friend function requires two parameters. `A + B` is interpreted globally as `operator+(A, B)`.

Example: Binary Plus (+) Operator (Friend Function)

- class Complex {
- int real, imag;
- public:
- Complex(int r=0, int i=0) : real(r), imag(i) {}
- friend Complex operator+(const Complex& lhs, const Complex& rhs);
- };
- Complex operator+(const Complex& lhs, const Complex& rhs) {
- return Complex(lhs.real + rhs.real, lhs.imag + rhs.imag);
- }

Example: Binary Plus (+) Operator (Friend Function)
class Complex {
int real, imag;
public:
Complex(int r=0, int i=0) : real(r), imag(i) {}
friend Complex operator+(const Complex& lhs, const Complex& rhs);
};
Complex operator+(const Complex& lhs, const Complex& rhs) {
return Complex(lhs.real + rhs.real, lhs.imag + rhs.imag);
}

Here is the friend version for adding Complex numbers. `operator+` takes two parameters, `lhs` and `rhs`. It explicitly accesses the real and imaginary parts of both objects. Since it's a friend, it can access private members of both passed objects.

Member Function vs. Friend Function

- When to use Member Functions:
- - When the left operand is ALWAYS an object of the class.
- - Mandatory for assignment =, subscript `[]`, function call `()`, and pointer access `->`.
- When to use Friend Functions:
- - When the left operand is a different type (e.g., adding an integer to an object: `5 + obj`).
- - Overloading stream operators << and >> (left operand is an ostream/istream object).

Member Function vs. Friend Function
- When to use Member Functions:
- - When the left operand is ALWAYS an object of the class.
- - Mandatory for assignment =, subscript [], function call (), and pointer access ->.
- When to use Friend Functions:
- - When the left operand is a different type (e.g., adding an integer to an object: 5 + obj).
- - Overloading stream operators << and >> (left operand is an ostream/istream object).

How do you choose? Generally, if the left side of the operator is your object, use a member function. Some operators MUST be member functions by C++ rules. However, if the left operand is a primitive type like `int` or a standard library class like `ostream` (for `cout << obj`), you MUST use a friend function because you cannot modify the external class to add a member function to it.

Best Practices & Common Pitfalls

- Do NOT change fundamental semantics (e.g., overloading + to perform subtraction confuses everyone).
- Return by value for arithmetic operators (+, -, *).
- Return by reference (`ClassType&`) for assignment (=) and compound assignment (+=) to allow chaining (`a = b = c`).
- Pass arguments by `const` reference (`const ClassType& obj`) to prevent unnecessary copying and accidental modification.
- Only overload operators if it genuinely makes the code more intuitive.

Best Practices & Common Pitfalls
- Do NOT change fundamental semantics (e.g., overloading + to perform subtraction confuses everyone).
- Return by value for arithmetic operators (+, -, *).
- Return by reference (ClassType&) for assignment (=) and compound assignment (+=) to allow chaining (a = b = c).
- Pass arguments by const reference (const ClassType& obj) to prevent unnecessary copying and accidental modification.
- Only overload operators if it genuinely makes the code more intuitive.

A few best practices: Never change the accepted mathematical meaning of a symbol. Pay attention to returns: arithmetic operators should yield new objects by value, but assignment operators should return a reference to allow assignment chaining. Always pass objects by `const` reference to save memory and processing time. Finally, don't abuse this feature; use it only when it adds clarity.

Summary & Q&A

- Operator overloading gives standard C++ operators custom functionality for objects.
- Implemented using the `operator` keyword.
- Member functions rely on the implicit `this` pointer (reducing argument count).
- Friend functions require all operands to be passed explicitly.
- Careful adherence to syntax, rules, and best practices ensures robust code.
- Any questions?

Summary & Q&A
- Operator overloading gives standard C++ operators custom functionality for objects.
- Implemented using the operator keyword.
- Member functions rely on the implicit this pointer (reducing argument count).
- Friend functions require all operands to be passed explicitly.
- Careful adherence to syntax, rules, and best practices ensures robust code.
- Any questions?

To summarize, operator overloading bridges the semantic gap between built-in primitive types and our custom classes, leading to much cleaner code. We explored the rules, the differences between member and friend functions, and best practices. In our next session, we will look into Type Conversions. Are there any questions?